# 谷歌 google.genai SDK 2025 工具调用（Function Call）深度实现与最佳实践报告

本报告针对谷歌最新的 google.genai SDK (2025 版本) 中的工具调用（Function Calling）功能，提供了一份详尽的技术分析、全面的代码示例以及生产级部署的注意事项和故障排除指南。本报告严格遵循最新的官方文档实现方法，并明确排除了已弃用的 Vertex AI SDK 模式 1。

## I. GenAI SDK 工具实现（2025 标准）概述

### A. 强制性转变：从 Vertex AI 到 google.genai

谷歌已将生成式 AI 功能的开发重心完全转移到独立的 google.genai SDK 上。原 Vertex AI SDK 中的生成式 AI 模块已被弃用，并将于 2026 年 6 月 24 日之后停止使用 1。因此，对于任何面向未来的应用开发，使用 google.genai 都是强制且必要的。

google.genai SDK 不仅仅是名称上的变更，它代表了一种架构上的现代化，提供了比旧版 SDK 更广泛的能力和更简化的接口 1。所有的代码示例都将使用 from google.genai import Client, types 进行初始化和配置，以确保符合最新的开发标准。

### B. 核心概念：工具、函数与执行边界

在 google.genai 生态系统中，“工具”（Tools）是模型可以访问并用来执行特定任务的外部接口。函数调用（Function Calling）是目前最核心的工具类型 3。

工具调用的基本机制遵循严格的职责分离原则：

1. **定义声明：** 开发者在应用代码中定义函数及其声明，向模型描述函数的作用、名称和参数 4。
2. **模型建议：** 模型分析用户请求、结合函数声明，决定是否需要调用工具。如果需要，模型会返回一个结构化的数据输出（一个 FunctionCall 对象），指定函数名和建议的参数 5。
3. **应用执行：** 核心原则是模型**绝不会执行代码本身** 4。开发者的应用程序负责解析模型返回的 FunctionCall 对象，并在应用层执行相应的本地函数或调用外部 API。

这个机制保证了工具调用的安全性和可控性，但要求开发者必须掌握多轮交互的执行流程（即下文详述的 4 步流程）4。

### C. 客户端初始化与环境配置

在使用 google.genai SDK 之前，必须正确初始化客户端。SDK 提供了同步（Sync）和异步（Async）两种客户端模式，适应不同的应用场景 6。

Python

import os  
from google.genai import Client  
from google.genai import types  
  
# 假设 GEMINI\_API\_KEY 已配置在环境变量中  
  
# 1. 同步客户端初始化 (推荐用于单线程应用)  
# 使用上下文管理器确保资源正确关闭  
try:  
 with Client() as client:  
 # 在此执行同步 API 调用  
 print("Synchronous Client Initialized.")  
 pass  
except Exception as e:  
 print(f"Client initialization failed: {e}")  
  
# 2. 异步客户端初始化 (推荐用于高性能、并发应用)  
async def initialize\_async\_client():  
 from google.genai import Client  
 # 使用.aio 属性访问异步客户端  
 async with Client().aio as aclient:  
 # 在此执行 await aclient.models.generate\_content(...) 调用  
 print("Asynchronous Client Initialized.")  
  
# 可以在主程序中通过 asyncio.run(initialize\_async\_client()) 调用

## II. 简化实现：自动函数调用（AFC）

自动函数调用（Automatic Function Calling, AFC）是 google.genai SDK 引入的一项重大简化功能，适用于大多数单轮工具使用场景，显著提高了开发效率 6。

### A. 运作原理：SDK 自动模式和 Schema 推理

在 AFC 模式下，开发者可以直接将 Python 函数本身作为工具传递给模型配置，而无需手动创建复杂的 FunctionDeclaration JSON Schema。

**SDK 模式的核心优势在于自动化 Schema 推理：** SDK 会自动检查所提供的 Python 函数的\*\*类型注解（Type Hints）\*\*和详细的 **Docstrings**，并据此构建模型所需的 OpenAPI 格式的函数声明 6。

**强制要求：** 为了让模型准确理解函数的作用和参数，Python 函数必须具备：

1. **精确的类型注解：** 定义参数和返回值的类型（例如 location: str -> str）。
2. **详细的 Docstrings：** 尤其是 Args: 部分，这是模型用来决定如何填充参数值的关键信息 7。模型无法查看函数体，其决策完全依赖于这些描述。

### B. 代码示例 1.0：基本同步自动工具使用

以下代码演示了如何定义一个 Python 函数并启用 AFC，实现端到端的自动执行和文本回复：

Python

from google.genai import Client, types  
  
# 1. 初始化客户端  
client = Client()  
  
# 2. 定义工具函数 (必须有明确的类型注解和详细的 Docstring)  
def get\_current\_weather(location: str, unit: str = "celsius") -> str:  
 """  
 返回指定地点和单位的当前天气。  
  
 Args:  
 location: 需要查询天气的城市和州/省，例如 "San Francisco, CA"。  
 unit: 温度单位，可以是 "celsius" 或 "fahrenheit"。默认为 "celsius"。  
 """  
 # 实际应用中，这里会调用外部 API  
 if "boston" in location.lower():  
 return f"Boston 当前天气为 20 {unit}，晴朗。"  
 else:  
 return "无法获取该地区的天气信息。"  
  
# 3. 调用模型并启用 AFC  
user\_prompt = "波士顿的天气怎么样？我需要知道摄氏度。"  
  
response = client.models.generate\_content(  
 model='gemini-2.5-flash',  
 contents=user\_prompt,  
 config=types.GenerateContentConfig(  
 # 直接传递 Python 函数对象  
 tools=[get\_current\_weather],  
 ),  
)  
  
# 4. 打印结果  
# 在 AFC 模式下，SDK 会在后台执行函数并返回最终的文本结果  
print("--- 自动函数调用结果 ---")  
print(response.text)

### C. 配置分析与 AFC 限制

AFC 机制的强大之处在于它将函数调用的 4 步流程（定义、调用、执行、响应）压缩成了一个单行 API 调用。types.GenerateContentConfig 中的 tools=[function\_name] 列表是启用该功能的关键配置。

尽管 AFC 极大地简化了开发，但其默认行为也存在限制：SDK 会自动处理工具的递归或顺序调用，但这种自动调用具有隐式限制 6。默认情况下，SDK 可能会在远程调用次数超过最大限制（默认为 10 次）时停止自动执行。

因此，对于涉及高度复杂、多步骤依赖或需要精细控制流程的“组合式函数调用”（Compositional Function Calling）任务，开发者需要退出 AFC 模式，转而采用手动、多轮的交互模式，以实现更高的控制力和稳定性。

| **Python 元素** | **结果 Schema 组成部分 (Gemini API)** | **模型感知功能的重要性** |
| --- | --- | --- |
| 函数名称 | function\_declaration.name | 必需，用于调用识别 |
| 函数 Docstring (摘要) | function\_declaration.description | 必需，用于模型选择使用场景 |
| Python 类型注解 (str, int) | 参数数据类型 (properties.type) | 必需，用于 JSON Schema 有效性 |
| Docstring (Args: 部分) | 参数描述 (properties.description) | 必需，用于模型精确选择参数值 |

## III. 权威实现：手动和顺序函数调用

手动函数调用模式提供了对流程的完全控制，是实现复杂、多步骤或“组合式”（Compositional）工作流的唯一可靠方法 4。

### A. 手动模式的必要性

以下场景要求开发者禁用 AFC，采用手动模式：

1. **组合式调用：** 当一个用户请求需要模型连续执行两次或多次工具调用，且后一次调用的输入依赖于前一次调用的结果时 4。
2. **自定义逻辑和错误处理：** 在函数执行前后需要插入自定义业务逻辑、日志记录或精细的错误捕获时。
3. **保持状态：** 在多轮对话中，需要明确控制每次 API 调用的内容历史，以确保上下文和性能优化（如利用思维签名）的准确性。

### B. 代码示例 2.0：禁用自动调用与提取建议

要切换到手动模式，必须在配置中显式禁用 AFC 6。

Python

from google.genai import Client, types  
  
client = Client()  
  
# 1. 定义工具函数 (与 AFC 相同，利用类型注解和 Docstrings)  
def list\_available\_stores(location: str) -> dict:  
 """返回指定地点附近所有可用商店的列表和 ID。"""  
 # 模拟返回 JSON 格式结果  
 return {"stores":, "location": location}  
  
# 2. 禁用 AFC 并进行首次调用  
user\_prompt = "帮我查一下旧金山有哪些商店？"  
  
response\_manual = client.models.generate\_content(  
 model='gemini-2.5-flash',  
 contents=user\_prompt,  
 config=types.GenerateContentConfig(  
 tools=[list\_available\_stores],  
 # 显式禁用自动调用，进入手动模式  
 automatic\_function\_calling=types.AutomaticFunctionCallingConfig(disable=True),  
 ),  
)  
  
# 3. 检查并提取函数调用建议  
function\_calls = response\_manual.function\_calls  
  
if function\_calls:  
 print("--- 模型已建议函数调用 (Manual Mode) ---")  
 print(f"建议调用的函数数量: {len(function\_calls)}")  
   
 # 解析第一个函数调用  
 first\_call = function\_calls  
 print(f"函数名称: {first\_call.name}")  
 print(f"参数: {dict(first\_call.args)}")  
   
 # 此时，应用程序需要执行该函数并将结果返回给模型  
else:  
 print("模型返回了文本响应:", response\_manual.text)

### C. 多轮工具调用循环（4 步流程）

手动模式要求开发者在应用程序中实现一个循环来管理模型、工具执行和上下文传递。这是实现组合式函数调用的核心。

1. **步骤 1: 定义函数声明**：如上所述，定义带有完整类型注解和 Docstrings 的 Python 函数。
2. **步骤 2: 调用模型并接收建议**：发送用户提示和函数声明给模型，并接收包含 FunctionCall 建议的响应。
3. **步骤 3: 应用层执行函数**：应用程序解析 FunctionCall，提取 name 和 args，并执行本地或外部工具代码 4。
4. **步骤 4: 发送结果并再次调用模型**：将函数执行结果封装成 FunctionResponse，附加到对话历史中，并作为新的用户回合（Role: user）内容，再次调用模型 4。模型将利用这个结果生成最终的用户友好响应。

### D. 代码示例 2.1：完整的组合式（顺序）工具调用循环

为了保持模型对上下文的理解并启用性能优化，必须将完整的对话历史（包括模型之前的响应内容）传递回模型。

Python

from google.genai import Client, types  
  
# 1. 初始化客户端  
client = Client()  
# 存储会话历史，包括用户提示、模型建议、函数结果  
conversation\_history =   
  
# 定义两个工具，实现顺序调用逻辑  
def get\_store\_id(store\_name: str) -> dict:  
 """根据商店名称查询其唯一的 ID。"""  
 return {"id": 102, "name": store\_name}  
  
def get\_store\_inventory(store\_id: int) -> dict:  
 """根据商店 ID 查询其最新的库存报告。"""  
 if store\_id == 102:  
 return {"store\_id": 102, "inventory": {"Apples": 50, "Bananas": 100}}  
 return {"error": "Store ID not found"}  
  
tools\_map = {  
 "get\_store\_id": get\_store\_id,  
 "get\_store\_inventory": get\_store\_inventory  
}  
  
# 核心：实现 4 步循环的函数  
def execute\_tool\_workflow(prompt: str, tools: list, max\_calls: int = 5):  
 global conversation\_history  
   
 # 第一次调用，或在循环中继续调用  
 conversation\_history.append(  
 types.Content(role="user", parts=[types.Part.from\_text(prompt)])  
 )  
   
 for \_ in range(max\_calls):  
 # 步骤 2: 调用模型  
 response = client.models.generate\_content(  
 model='gemini-2.5-flash',  
 contents=conversation\_history, # 传入完整的历史记录  
 config=types.GenerateContentConfig(  
 tools=tools,  
 automatic\_function\_calling=types.AutomaticFunctionCallingConfig(disable=True),  
 ),  
 )  
   
 # 将模型响应添加到历史记录 (即使是 FunctionCall 建议，也必须记录)  
 conversation\_history.append(response.candidates.content)  
   
 # 检查 FunctionCall  
 function\_calls = response.function\_calls  
   
 if not function\_calls:  
 # 步骤 4: 模型返回最终文本，流程结束  
 return response.text  
   
 # 步骤 3: 提取并执行函数  
 for call in function\_calls:  
 function\_name = call.name  
 function\_args = dict(call.args)  
   
 if function\_name in tools\_map:  
 print(f"-> 正在执行工具: {function\_name} with args: {function\_args}")  
   
 # 执行本地函数  
 result = tools\_map[function\_name](\*\*function\_args)  
   
 # 步骤 4: 构造 FunctionResponse Part 并添加到历史记录  
 function\_response\_part = types.Part.from\_function\_response(  
 name=function\_name,  
 response=result  
 )  
   
 # 将函数结果作为用户回合添加到历史，准备下一次 API 调用  
 conversation\_history.append(  
 types.Content(role="user", parts=[function\_response\_part])  
 )  
 else:  
 raise NotImplementedError(f"未知函数: {function\_name}")  
   
 return "达到最大调用限制，未能完成流程。"  
  
# 运行示例：模型首先需要调用 get\_store\_id，然后使用返回的 ID 调用 get\_store\_inventory  
final\_text = execute\_tool\_workflow(  
 prompt="Store A 的库存里有什么？",  
 tools=[get\_store\_id, get\_store\_inventory]  
)  
  
print("\n--- 最终用户响应 (Compositional Result) ---")  
print(final\_text)

**上下文管理与性能：** 在上述循环中，关键在于将 response.candidates.content 和 types.Content(role="user", parts=[function\_response\_part]) 持续添加到 conversation\_history 中。这种完整历史的传递模式自动包含了模型内部的\*\*“思维签名”（Thought Signatures）\*\* 4。思维签名是模型推理过程的加密表示，它在多轮对话中被自动传递，能够显著提升模型在后续工具调用决策中的稳定性和性能 4。

## IV. 高级工具模式与并行化

### A. 实现并行函数调用

并行函数调用是指模型在单个响应中返回多个独立的 FunctionCall 建议 4。这允许应用程序同时执行多个不相互依赖的工具，以节省时间。

**架构要求：** 为了高效处理并行调用，应用程序在解析 response.function\_calls 列表后，需要利用并发机制（如 Python 的 asyncio 或 concurrent.futures.ThreadPoolExecutor 8）来同时执行所有建议的函数。

**返回结构：** 在执行完毕后，所有的函数结果必须在下一个用户回合中，以单独的 types.Part.from\_function\_response 对象形式，一起返回给模型。

Python

# 假设我们在手动循环中接收到一个包含两个调用的 function\_calls 列表  
# [call\_1 (get\_weather), call\_2 (get\_time)]  
  
# 1. 并行执行 (概念示例，需集成并发库)  
results\_list =  
for call in function\_calls:  
 # 实际应用中，这里应使用线程池或 asyncio 调度执行  
 result = execute\_function\_sync(call.name, dict(call.args))   
 results\_list.append((call.name, result))  
  
# 2. 构造包含多个结果的单个用户回合  
parts\_to\_send =  
for name, result in results\_list:  
 parts\_to\_send.append(  
 types.Part.from\_function\_response(  
 name=name,  
 response=result  
 )  
 )  
  
# 3. 添加到历史并再次调用  
conversation\_history.append(types.Content(role="user", parts=parts\_to\_send))  
# client.models.generate\_content(..., contents=conversation\_history,...)

### B. 函数调用模式：控制模型行为

通过配置 types.FunctionCallingConfig，开发者可以显式控制模型的工具使用行为。

重点模式：ANY（强制工具使用）

当将模式设置为 ANY 时（例如：mode=types.FunctionCallingConfig.Mode.ANY），模型将被强制要求返回函数调用建议，即使它对使用该工具的信心不足 6。

这种模式对于需要进行结构化数据提取或强制执行 RAG（检索增强生成）查询的场景极为有用，因为它保证了模型的输出将是可操作的工具调用，而非纯文本。

### C. 代码示例 3.1：强制工具使用（ANY Mode）

Python

from google.genai import Client, types  
  
client = Client()  
  
def analyze\_sentiment(text: str) -> str:  
 """分析输入文本的情感倾向，返回 positive, negative 或 neutral。"""  
 return "neutral"  
  
# 即使提示词只是一个普通陈述句，模型也会被强制建议调用工具  
user\_prompt = "今天的会议进行得很好。"  
  
response\_forced = client.models.generate\_content(  
 model='gemini-2.5-flash',  
 contents=user\_prompt,  
 config=types.GenerateContentConfig(  
 tools=[analyze\_sentiment],  
 automatic\_function\_calling=types.AutomaticFunctionCallingConfig(disable=True),  
 # 强制模型尝试使用工具  
 tool\_config=types.ToolConfig(  
 function\_calling\_config=types.FunctionCallingConfig(  
 mode=types.FunctionCallingConfig.Mode.ANY  
 )  
 )  
 ),  
)  
  
print("--- 强制模式调用结果 ---")  
if response\_forced.function\_calls:  
 print(f"模型被强制建议调用函数: {response\_forced.function\_calls.name}")  
else:  
 print("意外：模型没有返回函数调用。")

| **工具模式** | **配置设定** | **主要用例** | **执行责任** | **控制级别** |
| --- | --- | --- | --- | --- |
| **自动 (默认)** | 直接传递 Python callable | 单轮快速原型、简单数据检索。 | SDK (自动执行和有限递归) | 低 (专注于最终文本输出) |
| **手动/顺序** | automatic\_function\_calling(disable=True) | 多轮决策路径、自定义错误处理、复杂工作流 (组合式调用)。 | 开发者 (提取建议、执行、重新调用) | 高 (完全控制 4 步循环) |
| **强制 (ANY)** | mode=types.FunctionCallingConfig.Mode.ANY | 保证输出工具建议；强制数据提取/RAG。 | 开发者 (提取建议) | 高 (保证工具调用) |

## V. 最佳实践与性能优化

### A. 设计健壮的工具函数

1. **精确的接口描述：** 模型的决策能力完全依赖于函数签名和 Docstrings 的质量 7。必须确保参数类型准确，且 Args: 部分提供了清晰、无歧义的描述。
2. **处理延迟：** 如果工具函数涉及高延迟的外部 API 调用，推荐在手动模式下使用异步客户端 (Client().aio) 和 asyncio 进行并发执行 3。高延迟的同步调用会显著降低用户体验。
3. **副作用与幂等性：** 理想情况下，工具应该设计为幂等（多次调用具有相同结果）或至少能够安全地处理重复调用。由于模型可能会基于不完整的上下文多次提出相同的调用建议，状态管理必须谨慎。

### B. 性能调优：上下文与思维签名

模型的生成式 API 是无状态的。这意味着在多轮对话中，模型不会自动保留上一个回合的内部推理过程。

为了克服这一限制并提高复杂工具调用场景的性能，系统利用了“思维签名”机制 4。思维签名是模型内部推理过程的加密表示，它作为模型响应的一部分被返回。

**最佳实践：** 开发者在实现手动或顺序调用循环时，必须始终将模型的**完整内容历史**（包括前一次模型的响应）作为 contents 参数传递回 API 4。通过遵循这种标准的历史管理模式，应用程序无需进行任何额外的代码更改，即可自动包含并利用思维签名，从而提高模型在工具调用决策中的稳定性和准确性。

### C. 安全与数据防护考量

由于模型只是建议参数，而应用程序负责执行代码，因此在执行阶段必须集成严格的安全检查：

1. **输入验证与净化：** LLM 建议的函数参数应被视为来自外部的、未经净化的输入。例如，如果工具是执行数据库查询，则必须对模型建议的字符串参数进行 SQL 注入防护或路径遍历检查。
2. **权限控制：** 工具函数访问的外部系统（数据库、内部 API 等）必须实施严格的权限和身份验证控制，确保 LLM 驱动的调用不会超出预定的安全边界。

## VI. 故障排除与常见问题解答 (FAQ)

本节总结了在使用 google.genai SDK 实现工具调用时，开发者最常遇到的问题及其解决方案。

### A. 问题：模型未能建议函数调用

**原因与解决：**

1. **Schema 定义不佳：** 模型可能无法理解工具的用途。检查 Python 函数的 Docstring 是否清晰完整，并确认类型注解准确 7。
2. **提示词不明确：** 确保用户提示明确或隐式地表明了需要使用工具解决问题。
3. **模型能力限制：** 确保使用的是支持工具调用功能的模型（例如 gemini-2.5-flash）。

高级调试：

使用 ANY 模式 (强制工具使用) 进行测试。如果模型在 ANY 模式下能返回调用建议，则问题在于模型的推理能力（即模型认为该工具不是最佳选择），而非 Schema 定义本身。

### B. 问题：工具参数解析错误或参数不正确

**原因与解决：**

1. **Docstring 歧义：** 这是最常见的问题。如果 Args: 部分的描述不精确，模型可能会误解所需的参数类型或含义。例如，避免使用模糊的术语，而应明确指定参数的格式（例如，"A valid 5-digit postal code string"）。
2. **非标准类型：** 尽量使用 JSON 兼容的基础类型 (str, int, float, list, dict)。如果需要更复杂的对象，可以考虑将输入参数设计为结构化的 JSON 字符串。

### C. 问题：TypeError: Invalid input type... Expected genai.ToolType

原因与解决：

这个 TypeError 通常发生在开发者试图将原始 Python 函数对象传递给需要正式 Tool 或 FunctionDeclaration 对象的配置区域时 10。

**解决方案：**

* **AFC 模式：** 仅在 config=types.GenerateContentConfig(tools=[callable]) 中可以直接传递原始 Python callable。
* **手动声明模式：** 如果需要手动定义 FunctionDeclaration，应该使用 types.FunctionDeclaration.from\_callable(callable=your\_function) 来显式创建声明 4，或使用 types.Tool(function\_declarations=[...]) 将其封装成一个 Tool 对象，并将其传递给配置。

### D. 调试手动模式下的函数调用部分

在手动模式下调试时，应使用以下标准模式来检查模型是否成功建议了调用：

Python

if response.function\_calls:  
 for function\_call in response.function\_calls:  
 print(f"DEBUG: Suggested Function Name: {function\_call.name}")  
 print(f"DEBUG: Suggested Arguments: {dict(function\_call.args)}")  
 # 确保应用程序工具映射 (tools\_map) 能够正确识别并执行该名称

这种调试方式能够明确区分是模型未能建议调用，还是应用程序未能正确解析和执行模型建议的调用。

## VII. 总结与建议

google.genai SDK 提供的工具调用能力为开发者开启了强大的应用扩展潜力。本次分析的核心结论是，该 SDK 建立了一个**双模式工具范式**，开发者需要根据使用场景选择合适的模式：

1. **自动模式 (AFC)：** 适用于快速原型开发和单轮、低复杂度的工具交互。利用 Python 类型注解和 Docstrings 即可实现，开发成本极低。
2. **手动/顺序模式：** 适用于需要完全控制流程、执行组合式（多步骤）任务，或在执行前后插入自定义业务逻辑的生产环境。尽管实现复杂度更高，但它提供了对性能优化（通过管理思维签名）和流程稳定性的最大控制。

**关键建议：**

* **架构选择：** 对于任何涉及 2 步以上工具调用的工作流，强烈推荐禁用 AFC，采用手动顺序调用循环。这能确保在面对复杂或边界情况时，流程具有可预测性和高鲁棒性。
* **代码规范：** 严格遵循 Python 类型注解和详细 Docstring 规范是所有工具调用的基础，无论采用何种模式，这将直接决定模型对工具的使用精度。
* **安全性前置：** 鉴于 LLM 建议参数的本质，应用层必须在执行任何带有副作用的工具函数之前，实现严格的参数验证和净化机制。

#### Works cited

1. Vertex AI SDK migration guide - Google Cloud Documentation, accessed on November 7, 2025, <https://docs.cloud.google.com/vertex-ai/generative-ai/docs/deprecations/genai-vertexai-sdk>
2. Good bye Vertex AI SDK - Medium, accessed on November 7, 2025, <https://medium.com/google-cloud/good-bye-vertex-ai-sdk-dcf90918239a>
3. Tool use with Live API | Gemini API - Google AI for Developers, accessed on November 7, 2025, <https://ai.google.dev/gemini-api/docs/live-tools>
4. Function calling with the Gemini API | Google AI for Developers, accessed on November 7, 2025, <https://ai.google.dev/gemini-api/docs/function-calling>
5. Function calling reference | Generative AI on Vertex AI - Google Cloud Documentation, accessed on November 7, 2025, <https://docs.cloud.google.com/vertex-ai/generative-ai/docs/model-reference/function-calling>
6. Google Gen AI SDK documentation, accessed on November 7, 2025, <https://googleapis.github.io/python-genai/>
7. Day 3 - Function calling with the Gemini API - Kaggle, accessed on November 7, 2025, <https://www.kaggle.com/code/markishere/day-3-function-calling-with-the-gemini-api>
8. Concurrent requests and generation - Gemini by Example, accessed on November 7, 2025, <https://geminibyexample.com/030-async-requests/>
9. Supercharging Gemini API Calls with Concurrency and a Custom Rate Limiter, accessed on November 7, 2025, <https://subhranil2605.medium.com/supercharging-gemini-api-calls-with-concurrency-and-a-custom-rate-limiter-2b021a85a3a2>
10. Function call not working · Issue #663 · google-gemini/deprecated-generative-ai-python, accessed on November 7, 2025, <https://github.com/google-gemini/generative-ai-python/issues/663>